avro

Avro Component

Available as of Camel 2.10

This component provides a dataformat for avro, which allows serialization and deserialization of messages using Apache Avro's binary dataformat. Moreover, it provides support for Apache Avro's rpc, by providing producers and consumers endpoint for using avro over netty or http.

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-avro</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

Apache Avro Overview

Avro allows you to define message types and a protocol using a json like format and then generate java code for the specified types and messages. An example of how a schema looks like is below.

```
{"namespace": "org.apache.camel.avro.generated",
 "protocol": "KeyValueProtocol",

 "types": [
     {"name": "Key", "type": "record",
      "fields": [
          {"name": "key",   "type": "string"}
      ]
     },
     {"name": "Value", "type": "record",
      "fields": [
          {"name": "value",   "type": "string"}
      ]
     }
 ],

 "messages": {
     "put": {
         "request": [{"name": "key", "type": "Key"}, {"name": "value", "type": "Value"} ],
         "response": "null"
     },
     "get": {
         "request": [{"name": "key", "type": "Key"}],
         "response": "Value"
     }
 }
}
```

You can easily generate classes from a schema, using maven, ant etc. More details can be found at the Apache Avro documentation.

However, it doesn't enforce a schema first approach and you can create schema for your existing classes. **Since 2.12** you can use existing protocol interfaces to make RCP calls. You should use interface for the protocol itself and POJO beans or primitive/String classes for parameter and result types. Here is an example of the class that corresponds to schema above:

```
package org.apache.camel.avro.reflection;

public interface KeyValueProtocol {
    void put(String key, Value value);
    Value get(String key);
}

class Value {
    private String value;
    public String getValue() { return value; }
    public void setValue(String value) { this.value = value; }
}
```

Note: Existing classes can be used only for RPC (see below), not in data format.

Using the Avro data format

Using the avro data format is as easy as specifying that the class that you want to marshal or unmarshal in your route.

```
    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
        <route>
            <from uri="direct:in"/>
            <marshal>
                <avro instanceClass="org.apache.camel.dataformat.avro.Message"/>
            </marshal>
            <to uri="log:out"/>
        </route>
    </camelContext>
```

An alternative can be to specify the dataformat inside the context and reference it from your route.

```
    <camelContext id="camel" xmlns="http://camel.apache.org/schema/spring">
         <dataFormats>
            <avro id="avro" instanceClass="org.apache.camel.dataformat.avro.Message"/>
        </dataFormats>
        <route>
            <from uri="direct:in"/>
            <marshal ref="avro"/>
            <to uri="log:out"/>
        </route>
    </camelContext>
```

In the same manner you can umarshal using the avro data format.

Using Avro RPC in Camel

As mentioned above Avro also provides RPC support over multiple transports such as http and netty. Camel provides consumers and producers for these two transports.

```
avro:[transport]:[host]:[port][?options]
```

The supported transport values are currently http or netty.

Since 2.12 you can specify message name right in the URI:

```
avro:[transport]:[host]:[port][/messageName][?options]
```

For consumers this allows you to have multiple routes attached to the same socket. Dispatching to correct route will be done by the avro component automatically. Route with no messageName specified (if any) will be used as default.

When using camel producers for avro ipc, the "in" message body needs to contain the parameters of the operation specified in the avro protocol. The response will be added in the body of the "out" message.

In a similar manner when using camel avro consumers for avro ipc, the requests parameters will be placed inside the "in" message body of the created exchange and once the exchange is processed the body of the "out" message will be send as a response.

Note: By default consumer parameters are wrapped into array. If you've got only one parameter, **since 2.12** you can use `singleParameter` URI option to receive it direcly in the "in" message body without array wrapping.

Avro RPC URI Options

Name	Version	Description
`protocolClassName`		The class name of the avro protocol.
`singleParameter`	2.12	If true, consumer parameter won't be wrapped into array. Will fail if protocol specifies more then 1 parameter for the message
`protocol`		Avro procol object. Can be used instead of `protocolClassName` when complex protocol needs to be created. One cane used #name notation to refer beans from the Registry
`reflectionProtocol`	2.12	If protocol object provided is reflection protocol. Should be used only with `protocol` parameter because for `protocolClassName` protocol type will be autodetected

Avro RPC Headers

Name	Description
`CamelAvroMessageName`	The name of the message to send. In consumer overrides message name from URI (if any)

Examples

An example of using camel avro producers via http:

```
        <route>
            <from uri="direct:start"/>
            <to uri="avro:http:localhost:{{avroport}}?protocolClassName=org.apache.camel.avro.generated.
KeyValueProtocol"/>
            <to uri="log:avro"/>
        </route>
```

In the example above you need to fill `CamelAvroMessageName` header. **Since 2.12** you can use following syntax to call constant messages:

```
        <route>
            <from uri="direct:start"/>
            <to uri="avro:http:localhost:{{avroport}}/put?protocolClassName=org.apache.camel.avro.generated.
KeyValueProtocol"/>
            <to uri="log:avro"/>
        </route>
```

An example of consuming messages using camel avro consumers via netty:

```
        <route>
            <from uri="avro:netty:localhost:{{avroport}}?protocolClassName=org.apache.camel.avro.generated.
KeyValueProtocol"/>
            <choice>
                <when>
                    <el>${in.headers.CamelAvroMessageName == 'put'}</el>
                    <process ref="putProcessor"/>
                </when>
                <when>
                    <el>${in.headers.CamelAvroMessageName == 'get'}</el>
                    <process ref="getProcessor"/>
                </when>
            </choice>
        </route>
```

Since 2.12 you can set up two distinct routes to perform the same task:

```
        <route>
            <from uri="avro:netty:localhost:{{avroport}}/put?protocolClassName=org.apache.camel.avro.generated.
KeyValueProtocol">
            <process ref="putProcessor"/>
        </route>
        <route>
            <from uri="avro:netty:localhost:{{avroport}}/get?protocolClassName=org.apache.camel.avro.generated.
KeyValueProtocol&singleParameter=true"/>
            <process ref="getProcessor"/>
        </route>
```

In the example above, get takes only one parameter, so `singleParameter` is used and `getProcessor` will receive Value class directly in body, while `putProcessor` will receive an array of size 2 with String key and Value value filled as array contents.